A NOTE ON THE PARTICLE FLOW FILTER/GENERATOR

Liu Yang
Division of Applied Mathematics
Brown University
Providence, RI 02912, USA
liu_yang@brown.edu

ABSTRACT

We shall analyze particle flow filter in this note.

1 INTRODUCTION

Particle flow filters are a family algorithms developed to sample from distributions of interest, for example, posterior. It's basically trying to do the same thing as MCMC, which has the following disadvantages:

- MCMC methods are almost always computationally expensive in high-dimensional state spaces.
- MCMC methods are highly sequential, which makes it hard for parallel computing.
- It's not convenient to draw independent samples from MCMC.

Particle flow filters tackle this problem by looking for an ODE or SDE governing the motion of particles, so that the distribution of the particles start from the source distribution, e.g. prior, and end up with the target distribution, e.g. posterior.

Once we find the ODE/SDE, we can have independent samples from the target distribution by:

- Step 1: independently sample from the source distribution as particles.
- Step 2: solve ODE/SDE with the starting point of each particle as the initial condition.

Note that there is no interactions between particles, therefore this particle flow approach is highly parallelizable.

Actually there are too many ways to transport from the source distribution to the target distribution. To get the governing ODE/SDE:

- We need to assume a trajectory of distributions, which start from the source distribution and end at the target distribution.
- And the ODE/SDE also yields a trajectory of distributions, given by Fokker-Planck Equation.
- The two trajectories should overlap, so we have an equation for the drift term and diffusion term in ODE/SDE.

Note that the trajectory is not unique. It's actually a good question to ask, what is the optimal trajectory, in what sense.

2 DETERMINISTIC PARTICLE FLOW FILTER

2.1 GENERAL FORMULATION

Consider log-homotopy based particle flow:

$$\log p(\boldsymbol{x}, t) = \log g(\boldsymbol{x}) + t \log h(\boldsymbol{x}) - \log K(t), \tag{1}$$

where $g(\boldsymbol{x})$ and $h(\boldsymbol{x})$ are both probability density functions, and $K(t)$ is the normalization term so that $p(\boldsymbol{x}, t)$ is a probability density function for all $t \in [0, 1]$. Note that $\log K(0) = 0$. In the context of Bayesian learning, g is the density of prior, h is the density of likelihood, so that $p(\cdot, 1)$ is the density of posterior.

For deterministic particle flow

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{v}(\boldsymbol{u}, t), \tag{2}$$

we have Fokker-Planck equation with zero diffusion:

$$\frac{\partial p(\boldsymbol{x}, t)}{\partial t} = -\nabla \cdot (\boldsymbol{v} p). \tag{3}$$

Then we have

$$\begin{aligned} \nabla \cdot (\boldsymbol{v} p) &= -p \frac{\partial \log p}{\partial t} \\ &= -p(\log h - \frac{\partial}{\partial t} \log K(t)) \end{aligned} \tag{4}$$

i.e.

$$\nabla \cdot \boldsymbol{v} + \boldsymbol{v} \cdot \nabla \log p = -\log h + \frac{\partial}{\partial t} \log K(t) \tag{5}$$

Further, if we assume the particle flow is divergence free, i.e.,

$$\nabla \cdot \boldsymbol{v} = 0, \tag{6}$$

then we have

$$\boldsymbol{v} \cdot \nabla \log p = -\log h + \frac{\partial}{\partial t} \log K(t) \tag{7}$$

If we construct a divergence free velocity field $\boldsymbol{v}$ satisfying Equation 7, then starting from density g, we will get to density gh with trajectory $p(\boldsymbol{x}, t)$.

2.2 FRED'S FORMULATION

In Fred's formulation (Daum, 2016), $\boldsymbol{v}$ is given by

$$\boldsymbol{v} = -\frac{\nabla \log p}{\|\nabla \log p\|^2} \log h \tag{8}$$

Which leads to

$$\frac{\partial}{\partial t} \log K(t) = 0 \tag{9}$$

Note that $\log K(0) = 0$, we have

$$\log K(t) = 0, \quad \forall t \in [0, 1] \tag{10}$$

i.e.

$$p(\boldsymbol{x}, t) = g(\boldsymbol{x}) h^t(\boldsymbol{x}). \tag{11}$$

And $g(\boldsymbol{x}) h^t(\boldsymbol{x})$ has to be a probability density function. This is too strong from my point of view. Also I cannot see why $\nabla \cdot \boldsymbol{v} = 0$ in the above formulation.

2.3 NECESSARY CONDITION FOR THE EXISTENCE OF DIVERGENCE FREE FLOW

Actually, if the velocity field is divergence free, p and f are limited to a very small family of distributions.

Note that the Fokker-Planck Equation has an equivalent form

$$\frac{d}{dt} \log p(\boldsymbol{u}(\boldsymbol{x}_0, t), t) = -\nabla \cdot \boldsymbol{v}(\boldsymbol{u}, t), \tag{12}$$

where $\boldsymbol{u}(\boldsymbol{x}_0, \cdot)$ is the trajectory of the particle start from $\boldsymbol{x}_0$. Since $\boldsymbol{v}$ is divergence free, we then have

$$\frac{d}{dt} \log p(\boldsymbol{u}(\boldsymbol{x}_0, t), t) = 0 \tag{13}$$

That is to say, the density p is constant along the particle trajectory.

3 Stochastic Particle Flow Filter

3.1 General Formulation

Consider stochastic particle flow

$$d\boldsymbol{u} = \boldsymbol{v}(\boldsymbol{u}, t)dt + \boldsymbol{\sigma}(\boldsymbol{u}, t)d\mathbf{W}(t), \tag{14}$$

where $\boldsymbol{v}$ is the drift term, $\boldsymbol{\sigma}$ is the spatial homogeneous diffusion term (for simplicity). We have Fokker-Planck equation:

$$\frac{\partial p(\boldsymbol{x}, t)}{\partial t} = -\nabla \cdot (\boldsymbol{v}p) + \nabla \cdot (\boldsymbol{Q}\nabla p), \tag{15}$$

where $\boldsymbol{Q} = \frac{1}{2}\boldsymbol{\sigma}\boldsymbol{\sigma}^T$ is the diffusion tensor. Then we have:

$$\begin{cases} \log p(\boldsymbol{x}, t) = \log g(\boldsymbol{x}) + t \log h(\boldsymbol{x}) - \log K(t) & \text{(i)} \\ \dfrac{\partial p(\boldsymbol{x}, t)}{\partial t} = -\nabla \cdot (\boldsymbol{v}p) + \nabla \cdot (\boldsymbol{Q}\nabla p) & \text{(ii)} \end{cases} \tag{16}$$

$$\text{(i)} \Rightarrow \frac{\partial \log p(\boldsymbol{x}, t)}{\partial t} = \log h(\boldsymbol{x}) - \frac{d \log K(t)}{dt}$$

$$\text{(ii)} \Rightarrow \frac{\partial \log p(\boldsymbol{x}, t)}{\partial t} = -\nabla \cdot \boldsymbol{v} - \boldsymbol{v} \cdot \nabla \log p + Tr[\boldsymbol{Q}H(\log p)] + \nabla \log p \cdot \boldsymbol{Q} \cdot \nabla \log p$$

where H is the Hessian matrix. So we have

$$\log h(\boldsymbol{x}) - \frac{d \log K(t)}{dt} = - \nabla \cdot \boldsymbol{v} - \boldsymbol{v} \cdot \nabla \log p + Tr[\boldsymbol{Q}H(\log p)] + \nabla \log p \cdot \boldsymbol{Q} \cdot \nabla \log p \tag{17}$$

3.2 Normalization Term

Now let's deal with the normalization term. Note that:

$$K(t) = \int g(x)h(x)^t dx \tag{18}$$

Therefore

$$\begin{aligned} \frac{d \log K(t)}{dt} &= \frac{1}{K(t)} \frac{dK(t)}{dt} \\ &= \frac{\int g(\boldsymbol{x})h(\boldsymbol{x})^t \log h(\boldsymbol{x}) d\boldsymbol{x}}{\int g(\boldsymbol{x})h(\boldsymbol{x})^t d\boldsymbol{x}} \\ &= \mathbb{E}_{p(\boldsymbol{x},t)} \log h(\boldsymbol{x}), \end{aligned} \tag{19}$$

where $\mathbb{E}_{p(\boldsymbol{x},t)}$ means expectation with density $p(\boldsymbol{x}, t)$.

Remark: $K(t)$ is the partition function of $p(\boldsymbol{x}, t)$, so the derivative $d \log K(t)/dt$ should be the $p(\boldsymbol{x}, t)$-expectation of coefficient of t in $\log p$, which is $\log(h)$ here.

3.3 Governing Equation

Combining the equations we have the governing equation for the dynamics:

$$\log h(\boldsymbol{x}) - \mathbb{E}_{p(\boldsymbol{x},t)} \log h(\boldsymbol{x}) = - \nabla \cdot \boldsymbol{v} - \boldsymbol{v} \cdot \nabla \log p + Tr[\boldsymbol{Q}H(\log p)] + \nabla \log p \cdot \boldsymbol{Q} \cdot \nabla \log p \tag{20}$$

Note the following:

- There is no normalization term in derivatives of $\log p$, i.e. we have exact analytic expression of $\nabla \log p$ and $H(\log p)$.
- $p(\boldsymbol{x}, t)$-expectation could be approximated empirically with particles at time t as samples of $p(\boldsymbol{x}, t)$. A numerical approach: solve $\boldsymbol{v}$ and $\boldsymbol{Q}$ until time t_k, then go to t_{k+1}.
- In general, solving this equation is not easy, if g and h are complicated. But maybe we can use techniques from deep learning.

3.4 Numerical Experiment

Using neural networks to solve $\boldsymbol{v}$.

- Prior: Gaussian centered at $(1, 1)$, var = 1.0.
- Likelihood: Gaussian centered at $(-1, -1)$, var = 1.0.
- Posterior: Gaussian centered at $(0, 0)$, var= 0.5.

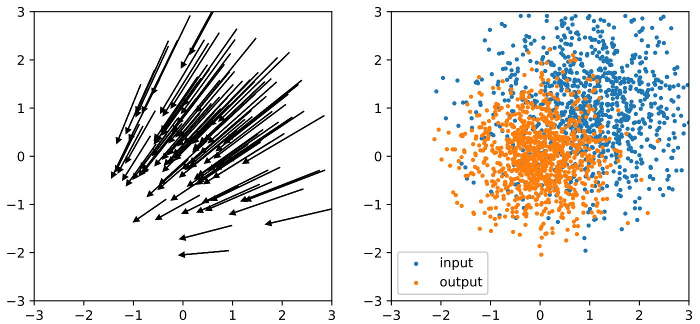

Figure 1: Left: map from $\boldsymbol{u}(0)$ to $\boldsymbol{u}(1)$. Right: samples of prior (blue) and posterior (orange). Mean of posterior samples: (-0.00346315, -0.00049885), variance of posterior samples: (0.49337277, 0.50254256). 10^6 samples.

3.5 Neglecting Normalization Term

In Drum Fred's formulation, the authors start from Equation 17, and take gradient of x to remove the normalization term $K(t)$, which is equivalent to

$$\begin{aligned}\log h(\boldsymbol{x}) + C(t) = & -\nabla \cdot \boldsymbol{v} - \boldsymbol{v} \cdot \nabla \log p + Tr[\boldsymbol{Q}H(\log p)] \\ & + \nabla \log p \cdot \boldsymbol{Q} \cdot \nabla \log p\end{aligned}$$

I actually got puzzled by the following paradox:

- If we neglect $K(t)$, then $p(\boldsymbol{x}, t)$ could be unnormalized, i.e. $\int_{\boldsymbol{x}} p(\boldsymbol{x}, t) d\boldsymbol{x} \neq 1$.
- On the other hand, $p(\boldsymbol{x}, t)$ represents the density of the particles. The particles never vanish, how could $\int_{\boldsymbol{x}} p(\boldsymbol{x}, t) d\boldsymbol{x} \neq 1$?

But later I figured out, it's about the vanishing condition of $\boldsymbol{v}p$ and $\boldsymbol{Q}\nabla p$ at infinity. Think about this 1D toy case:

$$p(x, 0) = \mathbf{1}_{x \geq 0} \frac{1}{(x+1)^2}$$

$$p(x, 1) = \mathbf{1}_{x \geq 0} \frac{1}{(2x+1)^2}$$

We can find a deterministic flow for p above ($x \to 2x$), but vp (flux) won't vanish at infinity. Actually, the time integral of flux converges to 0.5 as x goes to infinity, which means v goes to infinity.

Theoretically, it doesn't matter if p is normalized or not, since we only care about samples. But I think the explosion of v may lead to some numerical difficulty. So I would prefer not to neglect the normalization term.

4 Conclusion

Particle flow filter can be used to sample from (unnormalized) density. The time derivative of the normalization term could be formulated as an expectation, and be numerically tackled via sampling.

Solving the governing equation of particle flow is difficult in general, but it's possible to apply deep learning techniques. If we neglect the normalization term, theoretically it doesn't matter since we only care about samples. But the flux won't vanish at infinity, which might be dangerous from a numerical point of view.

REFERENCES

Fred Daum. seven dubious methods to compute optimal q for bayesian stochastic particle flow. In *2016 19th International Conference on Information Fusion (FUSION)*, pp. 2237–2244. IEEE, 2016.

APPENDIX

Code for the numerical experiment.

```
import logging, os
logging.disable(logging.WARNING)
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'


import argparse
import numpy as np
import tensorflow as tf
import random
import matplotlib.pyplot as plt

import sys



def feed_NN(X, W, b, act = tf.nn.tanh):
    A = X
    L = len(W)
    for i in range(L-1):
        A = act(tf.matmul(A, W[i]) + b[i])
    return tf.matmul(A, W[-1]) + b[-1]


def logPden_fun(x, dim = 2):
    sigma = 1.0
    logPden = - 0.5 * tf.reduce_sum((x - 1)**2/(sigma**2), axis = 1) \
              - 0.5 * dim * np.log(2*np.pi) - np.log(sigma) * dim
    return logPden

def logQden_fun(x, Qden, eps, dim = 2):
    sigma = 1.0
    logQden = - 0.5 * tf.reduce_sum((x + 1)**2/(sigma**2), axis = 1) \
              - 0.5 * dim * np.log(2*np.pi) - np.log(sigma) * dim
    return logQden


class PdataGenerator():
    def __init__(self, allsize = 40000):
        self.allsize = allsize
        self.alldata = (np.random.normal(0,1,[allsize,2])* np.array
    ([1,1])).astype(np.float32) + 1.0

    def nextbatch(self, bs):
        indexes = np.random.choice(self.allsize, bs, replace = False)
        return self.alldata[indexes,:]

    def nextnewbatch(self, bs):
```

```python
        return (np.random.normal(0,1,[self.allsize,2])* np.array([1,1])).astype(np.float32) + 1.0



def NN(x_in, dim_out, depth, width, act):
    A = x_in
    for _ in range(depth):
        A = tf.layers.dense(A, units= width, activation = act)
    A = tf.layers.dense(A, units= dim_out, activation = None)
    return A


def main(args):

    os.environ['CUDA_DEVICE_ORDER']='PCI_BUS_ID'    # see issue #152
    os.environ['CUDA_VISIBLE_DEVICES']= args.GPU

    tf.set_random_seed(args.random_seed)
    np.random.seed(args.random_seed)

    P = PdataGenerator()

    steps = args.steps; dt = 1.0 / steps
    Pdata = tf.placeholder(tf.float32, [None,2])



    if args.act == 'tanh':
        act = tf.nn.tanh
    elif args.act == 'lrelu':
        act = tf.nn.leaky_relu
    elif args.act == 'relu':
        act = tf.nn.relu
    elif args.act == 'softplus':
        act = tf.nn.softplus
    else:
        raise NotImplementedError

    v = [None for i in range(args.steps)]
    loss = [None for i in range(args.steps)]
    opt  = [None for i in range(args.steps)]
    us = [None for i in range(args.steps+1)]
    us[0] = Pdata

    if args.problem == 'Generative':
        for ti in range(0,args.steps):
            logPden = logPden_fun(us[ti])
            logQden = logQden_fun(us[ti], args.Qden, args.eps)
            gradlogp = (1-ti*dt) * tf.gradients(logPden, us[ti])[0]   \
                    + (ti*dt) * tf.gradients(logQden, us[ti])[0]
            with tf.variable_scope('v' + str(ti), reuse = tf.AUTO_REUSE):
                v[ti] = NN(us[ti], args.dim, args.nn_depth, args.nn_width, act)
            divv = tf.reduce_sum([tf.gradients(v[ti][:,i], us[ti])[0][:,i] for i in range(args.dim)], axis = 0)
            us[ti+1] = us[ti] + dt * v[ti]
            LHS = logQden - logPden - tf.reduce_mean(logQden - logPden, axis=0) # (batch_size,)
            RHS = - divv -  tf.reduce_sum(v[ti] * gradlogp, axis=1) # (batch_size,)
            loss[ti] = tf.reduce_mean((LHS - RHS)**2)
            var_list = [i for i in tf.trainable_variables() if 'v' + str(ti) in i.name]
```

```
            opt[ti] = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(loss[ti],
                                var_list=var_list)
            print(ti, end = ' ', flush=True)

    elif args.problem == 'Bayesian':
        for ti in range(0,args.steps):
            logPden = logPden_fun(us[ti])
            logQden = logQden_fun(us[ti], args.Qden, args.eps)
            gradlogp = tf.gradients(logPden, us[ti])[0]  \
                    + (ti*dt) * tf.gradients(logQden, us[ti])[0]
            with tf.variable_scope('v' + str(ti), reuse = tf.AUTO_REUSE):
                v[ti] = NN(us[ti], args.dim, args.nn_depth, args.nn_width, act)
            divv = tf.reduce_sum([tf.gradients(v[ti][:,i], us[ti])[0][:,i] for i in range(args.dim)], axis = 0)
            us[ti+1] = us[ti] + dt * v[ti]
            LHS = logQden - tf.reduce_mean(logQden, axis=0) # (batch_size,)
            RHS = - divv -  tf.reduce_sum(v[ti] * gradlogp, axis=1) # (batch_size,)
            loss[ti] = tf.reduce_mean((LHS - RHS)**2)
            var_list = [i for i in tf.trainable_variables() if 'v' + str(ti) in i.name]
            opt[ti] = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(loss[ti],
                                var_list=var_list)
            print(ti, end = ' ', flush=True)

    else:
        raise NotImplementedError


    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    sess = tf.Session(config=config)
    sess.run(tf.global_variables_initializer())

    step = 0

    savedir = 'save' + \
            '-Qden'+ args.Qden + \
            '-flow-'+ str(args.flow) + '-' + str(args.nn_depth) + 'x' + str(args.nn_width) + '-'+ str(args.act) + \
            '-seed' + str(args.random_seed)

    if not os.path.exists(savedir):
        os.mkdir(savedir)
    saver = tf.train.Saver(max_to_keep=1000)

    if args.restore > 0:
        saver.restore(sess, savedir+'/' + str(args.restore) + '.ckpt')
        P_test = P.nextnewbatch(1000000)
        thisloss = sess.run(loss, feed_dict = {Pdata: P_test})
        out = sess.run(us[-1], feed_dict={Pdata: P_test})
        print(step, np.sum(thisloss), np.mean(out, axis=0), np.var(out, axis=0), flush = True)

        plt.figure(figsize = (9,4))
        plt.subplot(1,2,1)
        thisP = P.nextbatch(args.batch_size)
        out = sess.run(us[-1], feed_dict={Pdata: thisP})
        for index in range(100):
            plt.arrow(thisP[index,0], thisP[index,1],
                        out[index,0] - thisP[index,0],
```

```
                        out[index,1] - thisP[index,1],
                        head_width=0.1,
                        head_length=0.1, color = 'k')
        plt.xlim((-3,3))
        plt.ylim((-3,3))

        plt.subplot(1,2,2)
        plt.scatter(thisP[:,0], thisP[:,1], s = 5, label = 'input')
        plt.scatter(out[:,0], out[:,1], s = 5, label = 'output')
        plt.legend()
        plt.xlim((-3,3))
        plt.ylim((-3,3))
        plt.savefig(savedir + '/' + str(args.restore)+'.eps', format = 'eps')
        return

    for ti in range(0,args.steps):
        for i in range(args.iterations + 1):
            if step % 500 ==0:
                saver.save(sess, savedir+'/' + str(i) + '.ckpt')
            if step % 100 ==0:
                plt.figure(figsize = (9,4))
                plt.subplot(1,2,1)
                thisP = P.nextbatch(args.batch_size)
                out = sess.run(us[-1], feed_dict={Pdata: thisP})
                for index in range(100):
                    plt.arrow(thisP[index,0], thisP[index,1],
                                out[index,0] - thisP[index,0],
                                out[index,1] - thisP[index,1],
                                head_width=0.1,
                                head_length=0.1, color = 'k')
                plt.xlim((-3,3))
                plt.ylim((-3,3))

                plt.subplot(1,2,2)
                plt.scatter(thisP[:,0], thisP[:,1], s = 5, label = 'input')
                plt.scatter(out[:,0], out[:,1], s = 5, label = 'output')
                plt.legend()
                plt.xlim((-3,3))
                plt.ylim((-3,3))

                plt.savefig(savedir + '/' + str(step)+'.png', dpi=50)


            # _, thisloss = sess.run([opt[ti], loss[ti]], feed_dict = {Pdata: P.nextbatch(args.batch_size)})
            # print(step, thisloss, flush = True)

            sess.run(opt, feed_dict = {Pdata: P.nextbatch(args.batch_size)})
            thisloss = sess.run(loss, feed_dict = {Pdata: P.nextbatch(args.batch_size)})
            out = sess.run(us[-1], feed_dict={Pdata: P.nextbatch(args.batch_size)})
            print(step, np.sum(thisloss), np.mean(out, axis=0), np.var(out, axis=0), flush = True)


            step += 1


if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='PDE Flow Generator')
```

```
    parser.add_argument('--dim', type = int, default=2,
                        help='space dimension')
    parser.add_argument('--GPU', type = str, default='0',
                        help='GPU index')
    parser.add_argument('-fl', '--flow', choices=['PDE'], default= 'PDE',
                        help='model')
    parser.add_argument('-rs','--random_seed', type = int, default= 0,
                        help='random seed for numpy and tensorflow')
    parser.add_argument('-bs','--batch_size', type = int, default= 1000,
                        help='batch size of training')
    parser.add_argument('--eps', type = float, default= 1e-4,
                        help='epsilon value used in log function')
    parser.add_argument('-nd','--nn_depth', type = int, default= 2,
                        help='number of hidden layers')
    parser.add_argument('-nw','--nn_width', type = int, default= 64,
                        help='width of hidden layers')
    parser.add_argument('--steps', type = int, default= 50,
                        help='steps of ODE flow')
    parser.add_argument('-it', '--iterations', type = int, default= 1000,
                        help='steps of ODE flow')
    parser.add_argument('--Qden', choices=['Eight1','Eight2','Nine'], default = 'Nine',
                        help='density of target distribution Q')
    parser.add_argument('--act', choices=['tanh','lrelu','relu','softplus'], default = 'tanh',
                        help='activation function for the neural net')
    parser.add_argument('-pr', '--problem', choices=['Bayesian','Generative'], default = 'Bayesian',
                        help='What problem are we solving?')
    parser.add_argument('-re', '--restore', type = int, default= 0,
                        help='steps to restore')
    args = parser.parse_args()

    main(args)
```